

# Principles of Programming Languages

<http://remus.rutgers.edu/cs314>

Topic: Introduction  
Professor Doug DeCarlo  
Spring 2004

## Course Goals

- **To gain an understanding of the basic structure of programming languages:**
  - Data types, control structures, naming conventions,...
- **To study different language paradigms:**
  - Functional (Scheme), Imperative (C), Object-Oriented (C++, Java), Logic (Prolog)
  - So that you can select an appropriate language for a task!
- **To learn the principles underlying all programming languages:**
  - So that it is easier to learn new languages!

# What is a programming language?

**“a language intended for use by a person to express a process by which a computer can solve a problem”**

**-Hope and Jipping**

**“a set of conventions for communicating an algorithm”**

**- E. Horowitz**

**“ the art of programming is the art of organizing complexity”**

**-Dijkstra, 1972**

# Why learn more than one PL?

- **Each language encourages you to think about a problem in a particular way**
- **Each language provides slightly different functionality**
  - **Look for a match between problem and language!**
- **Computer professionals should be multi-lingual:**
  - **Understand functionality and limitations of each PL**
  - **Specific applications sometimes require specialized PLs**
  - **PLs change over time as computer architecture changes**

# Imperative Paradigm

- **Paradigm: A sequence of state-changing actions**
- **Manipulate an abstract machine with:**
  - Variables that name memory locations
  - Arithmetic and logical operations
  - Reference, evaluate, assign operations
  - Explicit control flow statements
- **Fits the Von Neumann architecture closely**
- **Key operations: *Assignment* and “GoTo”**

# Imperative Paradigm

Sum up twice each  
number from 1 to N.

**Fortran**

```
SUM = 0  
DO 11 K=1,N  
SUM = SUM + 2*K  
11 CONTINUE
```

**C**

```
sum = 0;  
for (k = 1; k <= n; ++k)  
sum += 2*k;
```

**Pascal**

```
sum := 0;  
for k := 1 to n do  
sum := sum + 2*k;
```

# Functional Paradigm

- **Paradigm: Composition of operations on data**
- **Characteristics (in pure form):**
  - No named memory locations
  - Value binding through parameter passing
  - Recursion rather than iteration
- **Key operations: *Function Application* and *Function Abstraction***
  - Based on the Lambda Calculus

# Functional Paradigm

Scheme

```
(define (sum n)
  (if (= n 0)
      0
      (+ (* n 2) (sum (- n 1)))
  )
)
```

(sum 4) evaluates to 20

# Logic Paradigm

- **Paradigm: Formal logical specification of problem**
- **Characteristics (in pure form):**
  - Programs say *what* properties the solution must have, not *how* to find it
  - Solutions are obtained through a specialized form of *theorem-proving*
- **Key operations: *Unification* and *NonDeterministic Search***
  - **Based on First Order Predicate Logic**

# Logic Paradigm

```
sum(0,0).  
sum(N,S) :- NN is N - 1,  
           sum(NN, SS),  
           S is N * 2 + SS.
```

**Prolog**

```
?- sum(1,2).  
yes  
?- sum (2,4).  
no  
?-sum(20,S).  
S = 420  
?-sum (X,Y).  
X = 0 = Y
```

# Object-Oriented Paradigm

- **Paradigm:** Communication between abstract objects
- **Characteristics:**
  - “Objects” collect both the data and the operations
  - “Objects” provide *data abstraction*
  - Can be either imperative or functional (or logical)
- **Key operation:** *Message Passing* or *Method Invocation*

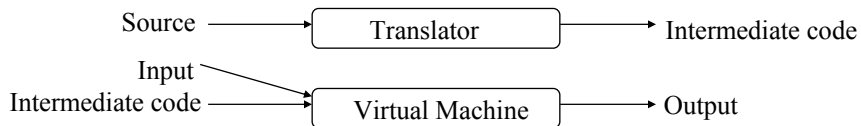
# Object-oriented Paradigm

```
class intSet : public Set
{ public: intSet() { }
//inherits Set add_element(), Set del_element()
//from Set class, defined as a set of Objects
  public int sum( ){
    int s = 0;
    SetEnumeration e = new SetEnumeration(this);
    while (e.hasMoreElements()) do
    {s = s + ((Integer)e.nextElement()).intValue();}
    return s;
  }
}
```

Java

# Translation

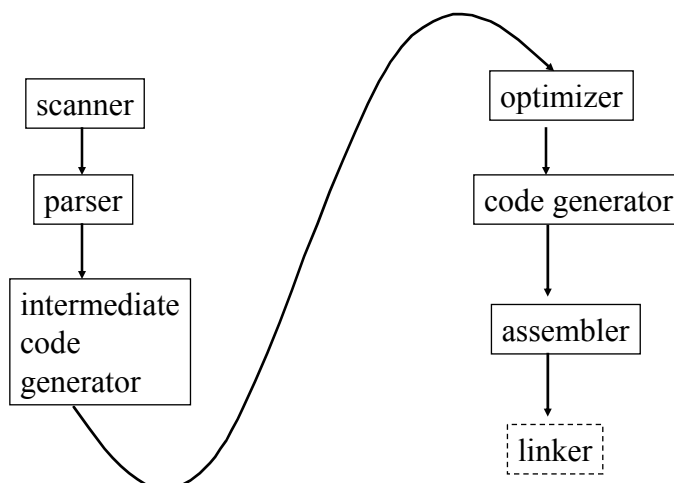
- **Compilation:** Program is translated from a high-level language into a form that is executable on an actual machine
- **Interpretation:** Program is translated and executed one statement at a time by a “virtual machine”
- **Most PL systems are a mixture of these two**
  - **Interpreted:** Java, Scheme, Prolog
  - **Compiled:** Fortran, C, C++



CS 314: Introduction

13

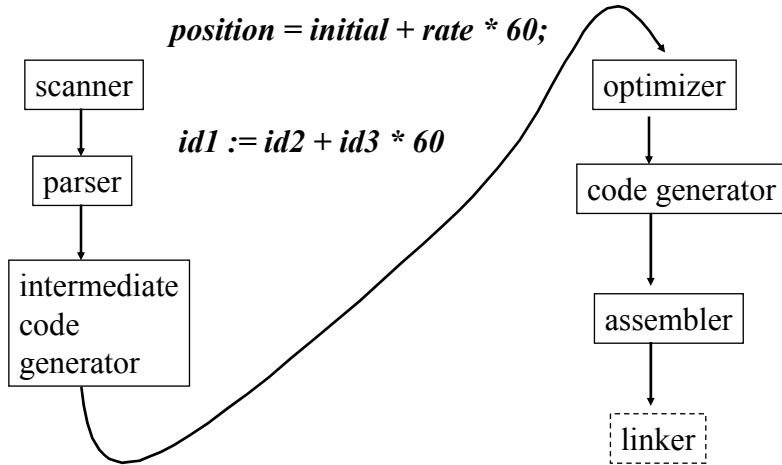
# Compilation



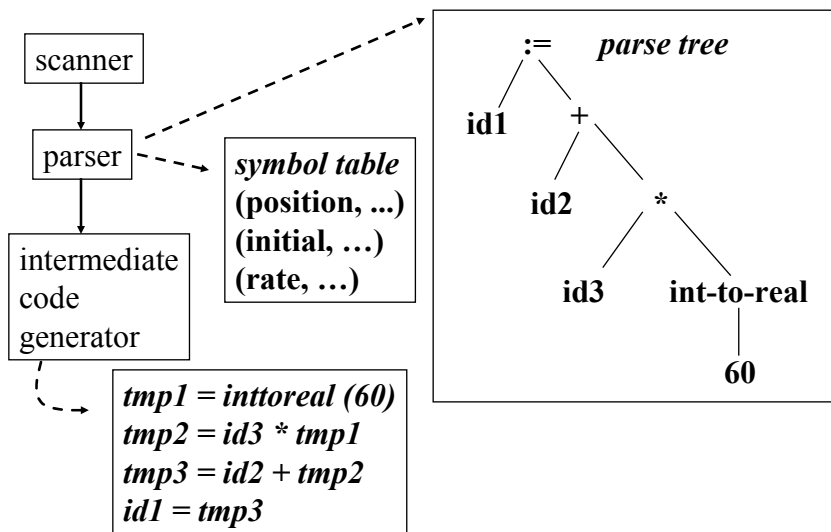
CS 314: Introduction

14

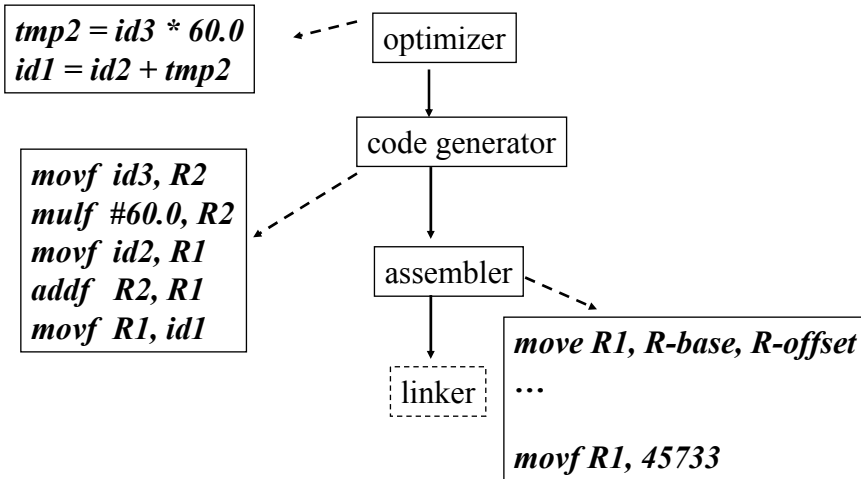
# Compilation



# Compilation



# Compilation



# History of PLs

- **Prehistory**
  - 300 B.C. Greece, Euclid invented the greatest common divisor algorithm - *oldest known algorithm*
  - ~1820-1850 England, Charles Babbage invented two mechanical computational devices
    - difference engine
    - analytical engine
    - Countess Ada Augusta of Lovelace, *first computer programmer*
  - **Precursors to modern machines**
    - 1940's United States, ENIAC developed to calculate trajectories

# History of PLs

- **1950's United States, first high-level PLs invented**
  - ***Fortran* 1954-57, John Backus (IBM on 704) designed for numerical scientific computation**
    - fixed format for punched cards
    - implicit typing
    - only counting loops, if test versus zero
    - only numerical data
    - 1957 optimizing Fortran compiler translates into code as efficient as hand-coded

# History of PLs

- ***Algol60* 1958-60, designed by international committee for numerical scientific computation [Fortran]**
  - block structure with lexical scope
  - free format, reserved words
  - while loops, recursion
  - explicit types
  - BNF developed for formal syntax definition
- ***Cobol* 1959-60, designed by committee in US, manufacturers and DoD for business data processing**
  - records
  - focus on file handling
  - English-like syntax

# History of PLs

- ***APL* 1956-60** Ken Iverson, (IBM on 360, Harvard) designed for array processing
  - functional programming style
- ***LISP* 1956-62**, John McCarthy (MIT on IBM704, Stanford) designed for non-numerical computation
  - uniform notation for program and data
  - new conditional control structure (COND)
  - recursion as main control structure
- ***Snobol* 1962-66**, Farber, Griswold, Polansky (Bell Labs) designed for string processing
  - powerful pattern matching

# History of PLs

- ***PL/I* 1963-66**, IBM designed for general purpose computing [Fortran, Algol60, Cobol]
  - user controlled exceptions
  - multi-tasking
- ***Simula67* 1967**, Dahl and Nygaard (Norway) designed as a simulation language [Algol60]
  - data abstraction
  - inheritance of properties
- ***Algol68* 1963-68**, designed for general purpose computing [Algol60]
  - orthogonal language design
  - interesting user defined types

# History of PLs

- ***Pascal* 1969, N. Wirth(ETH) designed for teaching programming [Algol60]**
  - 1 pass compiler
  - call-by-value semantics
- ***Prolog* 1972, Colmerauer and Kowalski designed for Artificial Intelligence applications**
  - theorem proving with unification as basic operation
  - logic programming
- **Recent**
  - ***C* 1974, D. Ritchie (Bell Labs) designed for systems programming**
    - allows access to machine level within high-level PL
    - efficient code generated

# History of PLs

- ***Clu* 1974-77, B. Liskov (MIT) designed for simulation [Simula]**
  - supports data abstraction and exceptions
  - precise algebraic language semantics
  - attempt to enable verification of programs
- ***Smalltalk* mid-1970s, Alan Kay (Xerox Parc), considered first real object-oriented PL, [Simula]**
  - encapsulation, inheritance
  - easy to prototype applications
  - hides details of underlying machine
- ***Scheme* mid-1970s, Guy Steele, Gerald Sussman (MIT)**
  - Static scoping and first-class functions

# History of PLs

- **Concurrent Pascal 1976** Per Brinch Hansen (U Syracuse) designed for asynchronous concurrent processing [Pascal]
  - monitors for safe data sharing
- **Modula 1977** N. Wirth (ETH), designed language for large software development [Pascal]
  - to control interfaces between sets of procedures or *modules*
  - real-time programming
- **Ada 1979**, US DoD committee designed as general purpose PL
  - explicit parallelism - *rendezvous*
  - exception handling and generics (*packages*)

# History of PLs

- **C++ 1985**, Bjarne Stroustrup (Bell Labs) general purpose
  - goal of type-safe object-oriented PL
  - compile-time type checking
  - templates
- **Java ~1995**, J. Gosling (SUN)
  - aimed at portability across platform through use of JVM - abstract machine to implement the PL
  - aimed to *fix* some problems with previous OOPLs
    - multiple inheritance
    - static and dynamic objects
  - ubiquitous exceptions
  - thread objects